Appendix C Solutions

C.1 a.

```
R1 LD    DADDI
R1 DADDI SD
R2 LD    DADDI
R2 SD    DADDI
R2 DSUB  DADDI
R4 BNEZ  DSUB
```

b. Forwarding is performed only via the register file. Branch outcomes and targets are not known until the end of the execute stage. All instructions introduced to the pipeline prior to this point are flushed.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
`LD R1, 0(R2)`	F	D	X	M	W													
`DADDI R1, R1, #1`		F	s	s	D	X	M	W										
`SD 0(R2), R1`					F	s	s	D	X	M	W							
`DADDI R2, R2, #4`								F	D	X	M	W						
`DSUB R4, R3, R2`									F	s	s	D	X	M	W			
`BNEZ R4, Loop`												F	s	s	D	X	M	W
`LD R1, 0(R2)`																	F	D

Since the initial value of R3 is R2 + 396 and equal instance of the loop adds 4 to R2, the total number of iterations is 99. Notice that there are 8 cycles lost to RAW hazards including the branch instruction. Two cycles are lost after the branch because of the instruction flushing. It takes 16 cycles between loop instances; the total number of cycles is 98 × 16 + 18 = 1584. The last loop takes two addition cycles since this latency cannot be overlapped with additional loop instances.

c. Now we are allowed normal bypassing and forwarding circuitry. Branch outcomes and targets are known now at the end of decode.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
`LD R1, 0(R2)`	F	D	X	M	W													
`DADDI R1, R1, #1`		F	D	s	X	M	W											
`SD R1, 0(R2)`			F	s	D	X	M	W										
`DADDI R2, R2, #4`					F	D	X	M	W									
`DSUB R4, R3, R2`						F	D	X	M	W								
`BNEZ R4, Loop`							F	s	D	X	M	W						
(incorrect instruction)									F	s	s	s	s					
`LD R1, 0(R2)`										F	D	X	M	W				

Again we have 99 iterations. There are two RAW stalls and a flush after the branch since the branch is taken. The total number of cycles is 9 × 98 + 12 = 894. The last loop takes three addition cycles since this latency cannot be overlapped with additional loop instances.

d. See the table below.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18
`LD    R1, 0(R2)`	F	D	X	M	W													
`DADDI R1, R1, #1`		F	D	s	X	M	W											
`SD    R1, 0(R2)`			F	s	D	X	M	W										
`DADDI R2, R2, #4`					F	D	X	M	W									
`DSUB  R4, R3, R2`						F	D	X	M	W								
`BNEZ  R4, Loop`							F	s	D	X	M	W						
`LD    R1, 0(R2)`									F	D	X	M	W					

Again we have 99 iterations. We still experience two RAW stalls, but since we correctly predict the branch, we do not need to flush after the branch. Thus, we have only 8 × 98 + 12 = 796.

e. See the table below.

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20
`LD    R1, 0(R2)`	F1	F2	D1	D2	X1	X2	M1	M2	W1	W2										
`DADDI R1, R1, #1`		F1	F2	D1	D2	s	s	s	X1	X2	M1	M2	W1	W2						
`SD    R1, 0(R2)`			F1	F2	D1	s	s	s	D2	X1	X2	M1	M2	W1	W2					
`DADDI R2, R2, #4`				F1	F2	s	s	s	D1	D2	X1	X2	M1	M2	W1	W2				
`DSUB  R4, R3, R2`					F1	s	s	s	F2	D1	D2	s	X1	X2	M1	M2	W1	W2		
`BNEZ  R4, Loop`									F1	F2	D1	s	D2	X1	X2	M1	M2	W1	W2	
`LD    R1, 0(R2)`										F1	F2	s	D1	D2	X1	X2	M1	M2	W1	W2

We again have 99 iterations. There are three RAW stalls between the LD and ADDI, and one RAW stall between the DADDI and DSUB. Because of the branch prediction, 98 of those iterations overlap significantly. The total number of cycles is 10 × 98 + 19 = 999.

f. 0.9ns for the 5-stage pipeline and 0.5ns for the 10-stage pipeline.

g. CPI of 5-stage pipeline: 796/(99 × 6) = 1.34.
CPI of 10-stage pipeline: 999/(99 × 6) = 1.68.
Avg Inst Exe Time 5-stage: 1.34 × 0.9 = 1.21.
Avg Inst Exe Time 10-stage: 1.68 × 0.5 = 0.84.

C.2 a. This exercise asks, "How much faster would the machine be . . .," which should make you immediately think speedup. In this case, we are interested in how the presence or absence of control hazards changes the pipeline speedup. Recall one of the expressions for the speedup from pipelining presented on page C-13.

$$\text{Pipeline speedup} = \frac{1}{1 + \text{Pipeline stalls}} \times \text{Pipeline depth} \quad \text{(S.1)}$$

where the only contributions to Pipeline stalls arise from control hazards because the exercise is only focused on such hazards. To solve this exercise, we will compute the speedup due to pipelining both with and without control hazards and then compare these two numbers.

For the “ideal” case where there are no control hazards, and thus stalls, Equation S.1 yields

$$\text{Pipeline speedup}_{\text{ideal}} = \frac{1}{1+0}(4) = 4 \quad \text{(S.2)}$$

where, from the exercise statement the pipeline depth is 4 and the number of stalls is 0 as there are no control hazards.

For the “real” case where there are control hazards, the pipeline depth is still 4, but the number of stalls is no longer 0. To determine the value of Pipeline stalls, which includes the effects of control hazards, we need three pieces of information. First, we must establish the “types” of control flow instructions we can encounter in a program. From the exercise statement, there are three types of control flow instructions: taken conditional branches, not-taken conditional branches, and jumps and calls. Second, we must evaluate the number of stall cycles caused by each type of control flow instruction. And third, we must find the frequency at which each type of control flow instruction occurs in code. Such values are given in the exercise statement.

To determine the second piece of information, the number of stall cycles created by each of the three types of control flow instructions, we examine how the pipeline behaves under the appropriate conditions. For the purposes of discussion, we will assume the four stages of the pipeline are Instruction Fetch, Instruction Decode, Execute, and Write Back (abbreviated IF, ID, EX, and WB, respectively). A specific structure is not necessary to solve the exercise; this structure was chosen simply to ease the following discussion.

First, let us consider how the pipeline handles a jump or call. Figure S.43 illustrates the behavior of the pipeline during the execution of a jump or call. Because the first pipe stage can always be done independently of whether the control flow instruction goes or not, in cycle 2 the pipeline fetches the instruction following the jump or call (note that this is all we can do—IF must update the PC, and the next sequential address is the only address known at this point; however, this behavior will prove to be beneficial for conditional branches as we will see shortly). By the end of cycle 2, the jump or call resolves (recall that the exercise specifies that calls and jumps resolve at the end of the second stage), and the pipeline realizes that the fetch it issued in cycle 2 was to the wrong address (remember, the fetch in cycle 2 retrieves the instruction immediately following the control flow instruction rather than the target instruction), so the pipeline reissues the fetch of instruction $i + 1$ in cycle 3. This causes a one-cycle stall in the pipeline since the fetches of

	Clock cycle					
Instruction	**1**	**2**	**3**	**4**	**5**	**6**
Jump or call	IF	ID	EX	WB		
$i + 1$		IF	IF	ID	EX	. . .
$i + 2$			*stall*	IF	ID	. . .
$i + 3$				*stall*	IF	. . .

Figure S.43 Effects of a jump of call instruction on the pipeline.

instructions after $i + 1$ occur one cycle later than they ideally could have. Figure S.44 illustrates how the pipeline stalls for two cycles when it encounters a taken conditional branch. As was the case for unconditional branches, the fetch issued in cycle 2 fetches the instruction after the branch rather than the instruction at the target of the branch. Therefore, when the branch finally resolves in cycle 3 (recall that the exercise specifies that conditional branches resolve at the end of the third stage), the pipeline realizes it must reissue the fetch for instruction $i + 1$ in cycle 4, which creates the two-cycle penalty.

Figure S.45 illustrates how the pipeline stalls for a single cycle when it encounters a not-taken conditional branch. For not-taken conditional branches, the fetch of instruction $i + 1$ issued in cycle 2 actually obtains the correct instruction. This occurs because the pipeline fetches the next sequential instruction from the program by default—which happens to be the instruction that follows a not-taken branch. Once the conditional branch resolves in cycle 3, the pipeline determines it does not need to reissue the fetch of instruction $i + 1$ and therefore can resume executing the instruction it fetched in cycle 2. Instruction $i + 1$ cannot leave the IF stage until *after* the branch resolves because the exercise specifies the pipeline is only capable of using the IF stage while a branch is being resolved.

Combining all of our information on control flow instruction type, stall cycles, and frequency leads us to Figure S.46. Note that this figure accounts for the taken/not-taken nature of conditional branches. With this information we can compute the stall cycles caused by control flow instructions:

$$\text{Pipeline stalls}_{\text{real}} = (1 \times 1\%) + (2 \times 9\%) + (1 \times 6\%) = 0.24$$

where each term is the product of a frequency and a penalty. We can now plug the appropriate value for Pipeline stalls$_{\text{real}}$ into Equation S.1 to arrive at the pipeline speedup in the “real” case:

$$\text{Pipeline speedup}_{\text{ideal}} = \frac{1}{1 + 0.24}(4.0) = 3.23 \quad \text{(S.3)}$$

	Clock cycle					
Instruction	**1**	**2**	**3**	**4**	**5**	**6**
Taken branch	IF	ID	EX	WB		
$i + 1$		IF	*stall*	IF	ID	. . .
$i + 2$			*stall*	*stall*	IF	. . .
$i + 3$				*stall*	*stall*	. . .

Figure S.44 Effects of a taken conditional branch on the pipeline.

	Clock cycle					
Instruction	**1**	**2**	**3**	**4**	**5**	**6**
Not-taken branch	IF	ID	EX	WB		
$i + 1$		IF	*stall*	ID	EX	. . .
$i + 2$			*stall*	IF	ID	. . .
$i + 3$				*stall*	IF	. . .

Figure S.45 Effects of a not-taken conditional branch on the pipeline.

Finding the speedup of the ideal over the real pipelining speedups from Equations S.2 and S.3 leads us to the final answer:

$$\text{Pipeline speedup}_{\text{without control hazards}} = \frac{4}{3.23} = 1.24$$

Control flow type	Frequency (per instruction)	Stalls (cycles)
Jumps and calls	1%	1
Conditional (taken)	$15\% \times 60\% = 9\%$	2
Conditional (not taken)	$15\% \times 40\% = 6\%$	1

Figure S.46 A summary of the behavior of control flow instructions.

Thus, the presence of control hazards in the pipeline loses approximately 24% of the speedup you achieve without such hazards.

b. We need to perform similar analysis as in Exercise C.2 (a), but now with larger penalties. For a jump or call, the jump or call resolves at the end of cycle 5, leading to 4 wasted fetches, or 4 stalls. For a conditional branch that is taken, the branch is not resolved until the end of cycle 10, leading to 1 wasted fetch and 8 stalls, or 9 stalls. For a conditional branch that is not taken, the branch is still not resolved until the end of cycle 10, but there were only the 8 stalls, not a wasted fetch.

$$\text{Pipeline stalls}_{\text{real}} = (4 \times 1\%) + (9 \times 9\%) + (8 \times 6\%) = .04 + .81 + .48 = 1.33$$

$$\text{Pipeline speedup}_{\text{real}} = (1/(1 + 1.33)) \times (4) = 4/2.33 = 1.72$$

$$\text{Pipeline speedup}_{\text{without control hazards}} = 4/1.72 = 2.33$$

If you compare the answer to (b) with the answer to (a), you can see just how important branch prediction is to the performance of modern high-performance deeply-pipelined processors.

C.3 a. 2ns + 0.1ns = 2.1ns

b. 5 cycles/4 instructions = 1.25

c. Execution Time = I × CPI × Cycle Time

Speedup = $(I \times 1 \times 7)/(I \times 1.25 \times 2.1) = 2.67$

d. Ignoring extra stall cycles, it would be: $I \times 1 \times 7/I \times 1 \times 0.1 = 70$

C.4 a. Calculating the branch in the ID stage does not help if the branch is the one receiving data from the previous instruction. For example, loops which exit depending on a memory value have this property, especially if the memory is being accessed through a linked list rather than an array. There are many correct answers to this.

```
LOOP:    LW     R1, 4(R2)     # R1 = r2->value
         ADD    R3, R3, R1    # sum = sum + r2->value
         LW     R2, 0(R2)     # r2 = r2->next
         BNE    R2, R0, LOOP  # while (r2 != null) keep
                              looping
```

The second LW and ADD could be reordered to reduce stalls, but there would still be a stall between the LW and BNE.

C.5 (Same as C.4)

C.6 (Same as C.4)

C.7 a. Execution Time = I × CPI × Cycle Time

Speedup = $(I \times 6/5 \times 1)/(I \times 11/8 \times 0.6) = 1.45$

b. $CPI_{5\text{-stage}} = 6/5 + 0.20 \times 0.05 \times 2 = 1.22$,

$CPI_{12\text{-stage}} = 11/8 + 0.20 \times 0.05 \times 5 = 1.425$

Speedup = $(1 \times 1.22 \times 1)/(1 \times 1.425 \times 0.6) = 1.17$

C.8–C.11 (Same as C.7)

C.12 a. 21 – 5 = 16 cycles/iteration

	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
`L.D F6, 0(R1)`	IF	ID	EX	MM	WB																
`MUL.D F4, F2, F0`		IF	ID	s	M1	M2	M3	M4	M5	M6	M7	MM	WB								
`LD F6, 0(R2)`			IF	s	ID	EX	MM	WB													
`ADD.D F6, F4, F6`					IF	ID	s	s	s	s	s	A1	A2	A3	A4	MM	WB				
`S.D 0(R2), F6`						IF	s	s	s	s	s	ID	s	s	s	EX	MM	WB			
`DADDIU R1, R1, #`												IF	ID	s	s	s	EX	MM	WB		
`DSGTUI`													IF	s	s	s	ID	EX	MM	WB	
`BEQZ`																	IF	ID	EX	MM	WB

b. (Same as C.12)

c. (Same as C.12)

C.13 (Same as C.12)

C.14 a. There are several correct answers. The key is to have two instructions using different execution units that would want to hit the WB stage at the same time.

b. There are several correct answers. The key is that there are two instructions writing to the same register with an intervening instruction that reads from the register. The first instruction is delayed because it is dependent on a long-latency instruction (i.e. division). The second instruction has no such delays, but it cannot complete because it is not allowed to overwrite that value before the intervening instruction reads from it.